3死锁

2018年10月31日　　13:39

-
- 死锁概述

一. 资源问题

1) 临界资源=要互斥访问的、不可抢占性资源，互斥≈不可共享

1. 可重用性资源和消耗性资源

1) 可重用性资源：可供用户重复使用多次的资源

i. 不可共享：每个单元只能分配给一个进程

ii. 使用顺序：请求资源、使用资源、释放资源

iii. 单元数目相对固定：运行期间不能创建/删除

(1) 请求和释放通常用系统调用。设备：request/release；文件：open/close；互斥：wait/signal

(2) 计算机系统中大多数资源都可重用

2) 可消耗性资源/临时性资源：由进程动态地创建/消耗的资源

i. 单元数目在程序运行期间不断变化

ii. 被进程不断创造而使单元数目增加（放入缓冲区）

iii. 由进程消耗，不返回给该资源类

(1) 如通信消息就是可消耗性资源，由生产进程创建，消费进程消耗

2. 可抢占性资源和不可抢占性资源

1) 可抢占性资源：如低优先级进程的处理机被高优先级进程抢占

(1) 又如内存紧张时被挂到外存，即内存被抢占

(2) 不会引起死锁

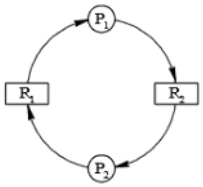
2) 不可抢占性资源：一旦被分配给进程，就只能被进程自行释放

(1) 如光盘刻录机、磁带机、打印机

二. 计算机系统中的死锁deadlock

1. 竞争不可抢占性资源

☑1) 如多进程请求打开多文件，资源分配图形成回路，说明已进入死锁

2) 

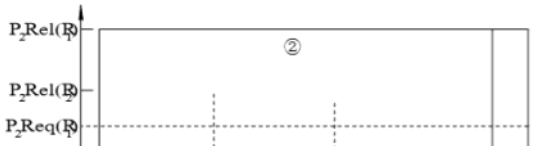
2. 竞争可消耗性资源

1) 如通信时都在等收到上家发送的消息，再向下家发送

3. 进程推进顺序不当

1) 合法：不会引起死锁的推进顺序

2) 非法：如四号路线进入了不安全区d，可能死锁



3)

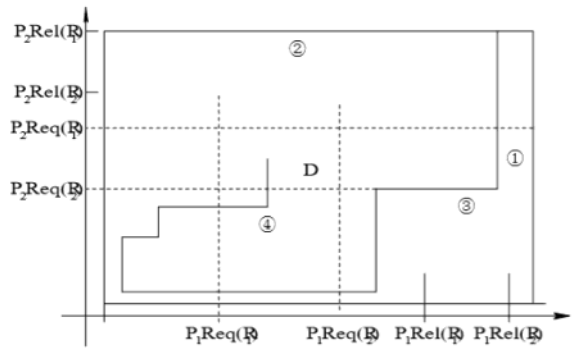


图 3-15　进程推进顺序对死锁的影响

三. 死锁的定义、必要条件、处理方法

1. **死锁：一组进程中各进程都在等待仅能由该组进程中其他进程引发的事件**
2. 产生死锁的**必要**条件：
   1) **互斥**：资源被其他进程占用，请求进程只能等待其释放
   2) **请求和保持**：进程已保持至少一个资源不放，还想请求新资源
   3) **不可抢占**：资源只在被进程使用完后释放
   4) **循环等待**：存在一个 进程-资源-进程-资源…… 的循环链
3. 处理死锁的方法
   (1) 预防：设置某些限制条件去破坏必要条件
   (2) 避免：资源分配过程中防止系统进入不安全状态
   (3) 检测：及时发现死锁的发生
   (4) 解除：发现死锁后立刻撤销部分进程
   1) 从上至下，预防程度弱，资源利用率高，阻塞频度低，并发程度高
      -
      - 预防死锁

一. 破坏“请求和保持”条件

1. 第一种协议：所有程序必须一次性申请完运行过程中可能需要的全部资源
   1) 一次分配完，破坏了请求；（申请不到全部就）不分配资源，破坏了保持
   2) 简单易行又安全，但
      (1) 资源利用率严重恶化：很多资源在运行时只用一会，被严重浪费
      (2) 进程饥饿现象：个别资源被占用使等待进程迟迟不能开始运行
2. 第二种协议：获得某一时期所需资源后便开始运行，边运行边释放资源
   1) 更快地完成任务，提高设备利用率，减少进程饥饿几率

二. 破坏“不可抢占”条件

1. 新资源请求不能满足时，释放已保持的所有资源，即可被其他进程抢占
2. 难实现，代价大，可能是前后两次运行的信息不连续，可能因反复申请释放导致进程执行被无限推迟，延长周转时间，增加系统开销，降低吞吐量

三. 破坏“循环等待”条件

1. “按序分配”：对资源类型做线性排序，赋予唯一序号，输入设备低，输出设备高；请求资源顺序必须按序号递增；想请求低序号类资源前，必须先释放同序号和高序号的资源。占据高序号资源的进程一定能不断推进
2. 资源利用率和系统吞吐量都得到改善，但

1) 资源排序限制了新类型资源的增加
2) 作业使用资源顺序与排序不同时会造成资源浪费
3) 限制了用户简单、自主的编程

-
- 避免死锁

一. 系统安全状态

1. 安全状态：能按某进程推进顺序为每个进程分配资源，使各进程都顺利完成
   1) 安全序列：↑这种推进顺序
   2) 不安全状态：无法找到安全序列的状态
   3) 不安全状态是死锁的必要非充分条件，安全是不死锁的充分非必要条件
2. 安全状态之例
   1)

进　程	最大需求	已 分 配	可　用
P_1	10	5	3
P_2	4	2	
P_3	9	2	

   2) 如图，<P2，P1，P3>即为安全序列，按此顺序能完成每个进程
3. 由安全状态向不安全状态的转换
   1) 不按照安全序列分配资源就可能会转换进不安全状态
   2) 资源分配前必须计算安全性，会进入不安全状态，则不分配

二. 银行家算法

1) Dijkstra为银行系统设计的，防止现金贷款时不能满足所有客户需求

1. 银行家算法中的数据结构
   1) 可利用资源向量Available：含有 m 个元素的数组
      (1) 每一个元素代表一类可利用的资源数目
      (2) 初值是系统中该类全部可用资源的数目，会动态地改变
      (3) Available[j]=K 表示系统中现有 Rj 类资源 K 个
   2) 最大需求矩阵 Max：n×m 的矩阵
      (1) 定义了 n 个进程中的每一个进程对 m 类资源的最大需求
      (2) Max[i][j]=K 表示进程 Pi 需要 Rj 类资源的最大数目为 K
   3) 分配矩阵 Allocation：n×m 的矩阵
      (1) 定义了系统中每一类资源当前已分配给每一进程的资源数
      (2) Allocation[i][j]=K 表示进程 Pi 当前已分得 Rj 类资源 K 个
   4) 需求矩阵 Need：n×m 的矩阵
      (1) 表示每一个进程尚需的各类资源数
      (2) Need[i][j]=K 表示进程 Pi 还需要 Rj类资源 K 个方能完成其任务
      (3) Need[i][j] = Max[i][j] - Allocation[i][j]

2. 安全性算法

1) 设置两个向量
   (1) 系统剩余可分配资源向量Work，有 m 个元素，初值同Available
   (2) 分配结束向量Finish，表示系统是否有足够的资源分配给进程，使之运行完成。初值全为false；有足够资源时，再令Finish[i]:=true
2) 寻找满足下述条件的进程

(1) Finish[i]=false;

(2) Need[i][j]≤Work[j];

3) 找到后分配资源，顺利完成后再全部释放

(1) Work[j] += Allocation[i][j];

(2) Finish[i] = true;

(3) go to step 2;

4) 如果第二步没找到，且存在Finish[i] = true，说明系统处于不安全状态

3．银行家算法

1) 设 Request i：进程 Pi的请求向量

(1) Request i[j]=K 表示进程 Pi 请求 K 个 Rj类型的资源

2) 发出资源请求后按下属步骤操作：

(1) 确认 Request i[j]≤Need[i,j]；不成立说明出错，所需资源数超过它声明的最大值

(2) 确认 Request i[j]≤Available[j]；不成立表示尚无足够资源，Pi 须等待

(3) 系统尝试分配资源，并修改下面的数值：

i. Available[j] -= Request i[j];

ii. Allocation[i][j] += Request i[j];

iii. Need[i][j] -= Request i[j];

(4) 执行安全性算法，确认安全才正式分配资源给进程 Pi，以完成本次分配；否则将本次的试探分配作废，恢复原状，让进程 Pi等待

4. 银行家算法之例

1) P1请求资源Request1(1，0，2)，系统按银行家算法进行检查

(1) Request1(1，0，2)≤Need1(1，2，2)

(2) Request1(1，0，2)≤Available1(3，3，2)

(3) 系统先假定可为 P1分配资源，并修改 Available，Allocation1和 Need1向量，由此形成的资源变化情况如图 3-16 中的圆括号所示

资源 / 情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	3	3	2
										(2	3	0)
P_1	3	2	2	2	0	0	1	2	2			
				(3	0	2)	(0	2	0)			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

图 3-16 T_0 时刻的资源分配表

(1) T0时刻的安全性：如图 3-17 ，利用安全性算法可知存在着安全序列{P1，P3，P4，P2，P0}，故系统是安全的，可按此序列分配资源（然而这个例子接下来演示的是如何作死，进入不安全序列）

资源 / 情况 / 进程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	3	2	1	2	2	2	0	0	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_2	7	4	5	6	0	0	3	0	2	10	4	7	true
P_0	10	4	7	7	4	3	0	1	0	10	5	7	true

图 3-17 T_0 时刻的安全序列

2) P4请求资源Request4(3，3，0)，系统按银行家算法进行检查

(1) Request4(3，3，0)≤Need4(4，3，1);

(2) Request4(3，3，0)≤Available(2，3，0)，让 P4等待

3) P0请求资源Requst0(0，2，0)，系统按银行家算法进行检查

(1) Request0(0，2，0)≤Need0(7，4，3);

(2) Request0(0，2，0)≤Available(2，3，0);

(3) 系统暂时先假定可为 P0分配资源，并修改有关数据，如图 3-19

进程 \ 资源 情况	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	3	2	2	1	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

图 3-19 为 P_0 分配资源后的有关资源数据

4) 进行安全性检查：Available(2，1，0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不再分配资源

-
- 死锁的检测与解除

一. 死锁的检测

1. 检测死锁需要：

1) 保存资源请求和资源分配信息

2) 提供算法检测是否已进入死锁状态

2. 资源分配图(Resource Allocation Graph)

1) 把结点集 N=PUR 划分为两个互斥的子集：

(1) 进程结点 P={p1，p2，…，pn}，一个圆圈表示一个进程或资源

(2) 资源结点 R={r1，r2，…，rn}，一个方块围住一些圈表示一类资源

(3) 在图 3-20 中，P={p1，p2}，R={r1，r2}， N={r1，r2}U{p1，p2}

(4)

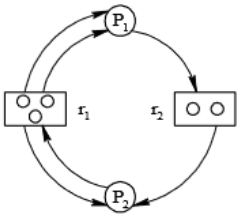


图 3-20 每类资源有多个时的情况

2) 对任意e∈E，都连接着 P 中的一个结点和 R 中的一个结点

(1) 资源请求边由进程指向资源，表示进程请求一个单位的资源

(2) 资源分配边由资源指向进程，表示把一个单位的资源分配给进程

3) 图 3-20 中，p1 进程已经分得了两个 r1 资源， 又请求一个 r2资源；p2 进程分得了一个 r1和一 个 r2资源，并又请求 r1资源

3. 死锁定理

1) 资源分配图的简化方法：

(1) 找出一个既不阻塞又非独立的进程结点 Pi。在顺利的情况下，Pi 可获得所需资源而运行完毕，再释放其所占有的全部资源，这相当于消去 pi所求的请求边和分配边，使之成为孤立的结点

(2)

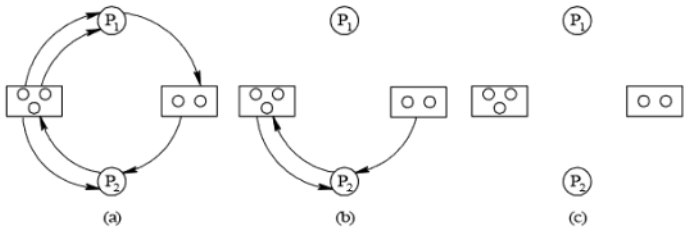


图 3-21　资源分配图的简化

(3) 循环：找下一个结点，并消去它的边

2) 若能消去图中所有的边，使所有的进程结点都成为孤立结点，则称该图是可完全简化的。可简化图无论按什么顺序都能得到同一个简化图

3) 死锁定理：死锁状态的充分条件是：资源分配图不可完全简化

4. 死锁检测中的数据结构

1) Available表示 m 类资源中每一类资源的可用数目

2) 把不占用资源的进程(向量 Allocationi:=0)记入 L 表中，即 LiUL

3) 从进程集合中找到一个 Requesti≤Work 的进程，做如下处理

(1) 简化资源分配图简化，释放资源，Work += Allocation I

(2) 将它记入 L 表中

4) 若不能把所有进程记入L表，说明将发生死锁

二. 死锁的解除

1. 解除方法：从某些进程抢占/剥夺足够资源，分配给死锁进程；终止/撤销某些进程，直至打破循环环路

1) 最简单的就是通知操作员人工处理

2) 另一类是用死锁解除算法：

2. 终止进程的方法

1) 终止所有死锁进程：简单，但代价很大，可能功亏一篑

2) 逐个终止进程，直至有足够资源

(1) 每终止一个进程，都要用死锁检测算法检查

(2) 每次要选择这些代价最小的进程终止：优先级、已执行时间、已使用资源、还需资源、交互式还是批处理式

3. 付出代价最小的死锁解除方法

1) 简单无脑但花费大的方法

(1) 先撤消进程 P1，使系统状态由 S→U1，付出的代价为 CU1

(2) 若仍处于死锁状态，需再撤消进程，直至解除死锁状态为止

(3) 再撤消进程 P2，使状态由 S→U2，其代价 为 CU2，…

(4) 可能付出的代价将是 k(k-1)(k-2)…/2C

2) 代价最小的方法：

(1) 先撤消一个死锁进程 P1，使系统状态由 S 演变成 U1

i. 将 P1记入 d(T)中，把付出代价 C1 记入 rc(T)中

ii. 对其他死锁进程重复上述过程，得到状态 U1，U2，…，Un

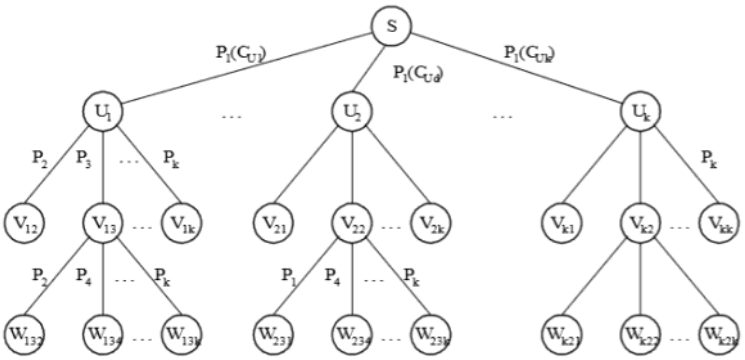
(2) 按代价大小，把进程插入到由 S 状态所演变的新状态的队列 L 中

i. 队列 L 中的首状态 U1 即为花费最小代价所演变成的状态

(3) 若仍处于死锁状态，再从 U1 状态按照上述处理方式再依次地撤消一个进程，得到 U′1，U′2，…，U′k 状态，再从 U′ 状态中选取一个代价最小的 U′j，直到死锁状态解除为止

(4) 花费代价：R(S)min = min{CUi} + min{CUj} + min{CUk} + …

(5)



i.

ii.

iii.

iv.

v.

vi.

vii. --------我是底线----------